

Take it EASEA

Pierre Collet, Evelyne Lutton, Marc Schoenauer, Jean Louchet

▶ **To cite this version:**

Pierre Collet, Evelyne Lutton, Marc Schoenauer, Jean Louchet. Take it EASEA. PPSN VI, Sep 2000, Paris - France. inria-00000875

HAL Id: inria-00000875
https://inria.hal.science/inria-00000875

Submitted on 29 Nov 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Category : Genetic Algorithms

Take it EASEA

Pierre COLLET, Evelyne LUTTON,
Projet Fractales — INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay cedex, France
Pierre.Collet@inria.fr, Evelyne.Lutton@inria.fr — Tel : +33 (0)1 39.63.55.52
`http://www-rocq.inria.fr/fractales`

Marc SCHOENAUER
EAAX – CMAPX — École Polytechnique, 91128 Palaiseau cedex, France
Marc.Schoenauer@polytechnique.fr
`http://www.eeaax.polytechnique.fr`

Jean LOUCHET
AMI – LEI — École Nationale Supérieure de Techniques Avancées,
35 Boulevard Victor, 75011 PARIS, France
Louchet@ensta.fr
`http://www.ensta.fr/~louchet`

January 24, 2000

Abstract

Evolutionary algorithms are not straightforward to implement and the lack of any specialised language forces users to reinvent the wheel every time they want to write a new program. Over the last years, evolutionary libraries have appeared, trying to reduce the amount of work involved in writing such algorithms from scratch, by offering standard engines, strategies and tools. Unfortunately, most of these libraries are quite complex to use, and imply a deep knowledge of object programming and C++. To further reduce the amount of work needed to implement a new algorithm, without however throwing down the drain all the man-years already spent in the development of such libraries, we have designed EASEA (acronym for **EA**sy **S**pecification of **E**volutionary **A**lgorithms): a new high-level language dedicated to the specification of evolutionary algorithms. EASEA compiles `.ez` files into C++ object files, containing function calls to a chosen existing library. The resulting C++ file is in turn compiled and linked with the library to produce an executable file implementing the evolutionary algorithm specified in the original `.ez` file.

EASEA v0.3 is available on the web at: `http://www-rocq.inria.fr/EVO-Lab/` .

1 Introduction

Not so long ago, evolutionary algorithms were considered as mere fantasies set up by mad computer scientists. No respectable researcher would ever have considered using such algorithms to do anything serious. Things have changed however over the years and many end-users (chemists, physicists, mathematicians, ...) have ended up selling their scientific souls to DARWIN. Unfortunately taking this decision is not the hardest part of their ordeal: the evolutionary algorithm they have been dreaming of remains to be written and many of them are only occasional programmers, used to procedural languages such as PASCAL or FORTRAN. This is very understandable as they are not state of the art computer scientists after all.

One way to speed up the process is to use one of the many existing evolutionary libraries. All is for the best as they offer very powerful tools provided ... one is fluent enough with constructors, copy-constructors, destructors and such niceties involved by relatively low-level object languages.

The next hurdle is then to learn how to use the library, to understand the intricate data structures and to memorise the necessary several hundred object types, functions and variables and the way they are inter-related. This can be quite time consuming when all major evolutionary libraries are written in C++ and make full use of object programming.

All in all, many physicists, chemists, mathematicians and other scientists who otherwise would be capable of writing relatively simple functions in C, FORTRAN or LISP are denied experimentation of evolutionary algorithms due to the sheer complexity of their implementation.

The aim of EASEA (**EA**sy **S**pecification of **E**volutionary **A**lgorithms) is to hide this complexity behind a high-level language, allowing scientists to concentrate on evolutionary algorithms, rather than on their implementation.

2 Previous work

Some research teams have already felt the need for a specific evolutionary language. They have however chosen a theoretic viewpoint, trying to enrich the evolutionary paradigm with new concepts or features not yet implemented [6, 7, 9, 10].

We have chosen a radically different approach, trying to be as pragmatic as possible. Our goal was to start with the realisation of a minimal working prototype, able to implement almost any problem. We count on feedback from end-users to guide the evolution of EASEA.

3 Presentation of EASEA

3.1 Introduction

Several important ideas lie behind the EASEA language and compiler :

- EASEA must be general enough to be able to write virtually any evolutionary algorithm.
- Conceptually speaking, a language such as EASEA needs not be tied to a specific evolutionary library. Hence, EASEA must be able to operate different evolutionary libraries.
- EASEA should aim to hide away all programming mechanisms not explicitly needed to describe the evolutionary algorithm.
- EASEA source files must be simple enough to be written automatically by a graphic user interface.

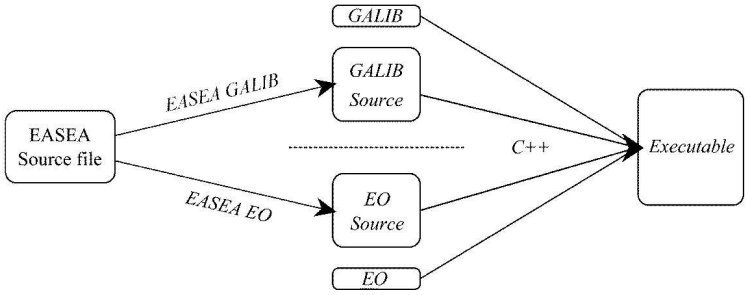


Figure 1: EASEA mode of operation

3.2 Mode of operation

The specifications of EASEA show that an EASEA compiler should be able to produce C++ source code using different evolutionary libraries.

Two libraries have been chosen to start with: GAlib —a widely used C++ genetic library [5]— and EO (Evolutionary Objects [3]) developed at the University of Granada (Spain) within the EVONET [1] framework.

EASEA–EO is still under development while EASEA–GAlib is already operational. The EASEA–GAlib compiler uses for input an ascii file with a `.ez` suffix. Its output is a GAlib C++ source file including calls to GAlib functions and objects. The resulting C++ file must then be compiled by a C++ compiler and linked with the GAlib library (cf. figure 1). The produced executable implements the evolutionary algorithm described in the original EASEA source file.

4 EASEA compiler

4.1 Description

EASEA is written in C++, using Lex and Yacc (in fact ALex and AYacc [4]). The EASEA compiler is somewhat unusual in the sense that it produces source code in another language rather than microprocessor instructions. As EASEA syntax is rather simple, most serious errors come in fact from user-functions which are compiled by a host C++ compiler. The nice consequences are that such errors are trapped by the very elaborate host compiler syntax analyser and that semantic errors (bugs) are as elaborately dealt with by the host compiler symbolic debugger. The not so nice consequence is that the human end-user must somehow debug the C++ code produced by EASEA.

The main difficulty resides in the fact that humans usually find compiler-produced source code quite difficult to read.

A second reason voting for highly readable generated C++ is that we think EASEA can also be used as a primer: EASEA creates a C++ source file which can be a starting point for an evolutionary algorithm that will be refined afterwards.

Our main concern has then been to improve presentation and to have EASEA-generated C++ look as human as possible.

This feat is mainly achieved through :

1. a man-made *template* file —`galib.tpl` for the GAlib version. As one can infer by its name, the GAlib template file contains the framework of a generic GAlib evolutionary algorithm, ready to be filled up with user-specific information found in the EASEA `.ez` source file.
2. very carefully typeset code, whenever EASEA generates code to fill up the blanks : indentation is respected, meaningful variable names are used and comments are generated from scratch to explain what the created code is supposed to do.

The compiler contains two main parts: one responsible for the genome analysis and the other responsible for code production.

4.2 Genome declaration analysis

EASEA genome declarations look very much like C or C++ structure declarations. `char`, `int`, `double`, `bool` are accepted as basic types. Modifiers are accepted, allowing arrays and pointers to be declared. Finally, it is possible to declare new types (classes in fact, as EASEA is fully object-oriented).

Let us imagine, as a demonstrative example, a genome representing a polygon:

```
Side{   int  Coord[2];
        Side *pNext;
}
Genome{ Side *pList;
        int  NbSides;
}
```

This genome is made of a pointer towards a linked list of sides and of `NbSides`, an integer containing the number of elements in the linked list.

A side is made of an array of two coordinates and a pointer towards the next side.

All variables are stored in a symbol table, along with their type and size (arrays). New user types are stored along with the elements they contain.

4.3 Generation of complete C++ classes

The template file is in fact an empty shell, containing the source code for a generic GALib evolutionary algorithm. The EASEA compiler copies lines from the template file towards the object `.cpp` file until it comes across a compiler directive telling it to insert information which is to be found in the user-supplied `.ez` file.

The user-defined types and functions are then inserted in the output `.cpp` file, as well as the genome declaration:

- New types are inserted as new C++ classes, with all methods necessary to obtain fully fledged C++ classes (constructor, destructor, copy-constructor, operator=, operator==, operator!=, operator<<, operator>>).

  Here is for instance the operator= member function, transparently created by the EASEA compiler for class `Side`:

  ```
  Side operator=(Side &EASEA_Var) {  // Operator=
    if (pNext) delete pNext;
    EASEA_Var.pNext ? pNext = new Side(*(EASEA_Var.pNext)) : pNext=NULL;
    {for(int EASEA_Ndx=0; EASEA_Ndx<2; EASEA_Ndx++)
       Coord[EASEA_Ndx]=EASEA_Var.Coord[EASEA_Ndx];}
    return *this;
  }
  ```

  - The `Coord` array has been automatically expanded and each of its elements are assigned individually.
  - If the R-value `pNext` pointer is not null (`EASEA_Var.pNext`), a new `Side` object is created, and its copy-constructor is invoked with the object pointed to by the R-value `pNext` pointer. This new `Side` object is assigned to the L-value `pNext` pointer.

    As an intelligent copy-constructor has also been created automatically for class `Size`, this results in a recursive copy of the linked list.

- The genome is derived from the GAlib genome class. As for new user classes, all necessary methods are transparently created.

  Here is for instance the derived GAlib `polygonGenome` class declaration, created out of the previous `Genome` declaration:

  ```
  // User Genome
  class polygonGenome : public GAGenome {
  // Default methods for class polygonGenome
  public:
    GADefineIdentity("polygonGenome", 251);
    static void Initializer(GAGenome&);
    static int Mutator(GAGenome&, float);
    static float Comparator(const GAGenome&, const GAGenome&);
    static float Evaluator(GAGenome&);
    static int Crossover(const GAGenome&, const GAGenome&, GAGenome*,
  GAGenome*);
  ```

```
  polygonGenome::polygonGenome() :GAGenome(Initializer, Mutator,
Comparator){
    evaluator(Evaluator); crossover(Crossover);
//   Zeroing pointers
    pList=NULL;
  }
  polygonGenome(const polygonGenome & orig) { copy(orig); }
  polygonGenome operator=(const GAGenome &);
  virtual GAGenome *clone(GAGenome::CloneMethod) const ;
  virtual void copy(const GAGenome & c);
  virtual int equal(const GAGenome& g) const;
  virtual int read(istream & is);
  virtual int write(ostream & os) const ;

// Class members
  int NbSides;
  Side *pList;
};
```

Of course, as for new types, all generic member functions are created transparently (constructor, destructor, copy-constructor, operator=), as well as all member functions required by GAlib (`clone`, `copy`, `equal`, `read`, `write`, `Comparator`).

The remaining member functions required by GAlib (`Initializer`, `Mutator`, `Evaluator` and `Crossover`) are, (in v0.3), user-specific. This means that EASEA will look for their code in the `.ez` file, under the names of `Genome::initializer`, `Genome::mutator`, `Genome::crossover`, `Genome::evaluator`, as can be seen in the extensive example of section 5. Future versions of EASEA will implement representation-independent operators in the line of Radcliffe's work [7].

In v0.3, EASEA expects to find C++ source code, that will be directly inserted in the resulting `.cpp`, qualified by minor changes described in the next section.

4.4 Remaining independent of host libraries

Accepting different host libraries means that EASEA cannot force the programmer to use the GAlib-specific `GARandomDouble` function. This would mean that an EASEA `.ez` source file would not recompile if another library were to be used.

The solution is for EASEA to provide (as in any new language), its own specific predefined functions and keywords. Whenever such functions or keywords appear in the `.ez` source file, the EASEA compiler translates them to their GAlib (or EO) equivalent. As such, the EASEA `random` function call will be translated in its GALib equivalent: `GARandomDouble`.

Identically, EASEA keywords are translated into their host-library equivalent. For example, variables `parent1`, `parent2`, `child1` and `child2` are predefined in the EASEA crossover function (cf. section 5). Those names are translated into their GAlib counterparts (`mom`, `dad`, `bro` and `sis`) ... if the GAlib library is to be used, of course.

This is how EASEA source files can try to be as independent as possible from host libraries.

4.5 Host libraries prerequisites

Although EASEA source files can conceptually be completely decoupled from the underlying host library, this is not the case right now. GAlib expects functions such as `mutator`, `evaluator` and `crossover` to behave a certain way: the `evaluator` function should return a `float`, the `crossover` function should return the number of created children (one or two), the `mutator` is supposed to return the number of mutations which occurred in the genome.

Unfortunately, it is very improbable that other libraries should have exactly the same expectations.

For the moment being, the EASEA manual describes how the GAlib library expects genetic operators to behave.

In the future, one can expect EASEA to impose its own prerequisites. If they are a superset of all the information needed by the different supported host libraries, EASEA will be able to generate the appropriate code implementing the behaviour expected by each library.

4.6 User-space for external functions and variables

Up to now, the end-user has been offered to modify only four genome-related functions. This might not prove sufficient if additional functions or external variables are needed. Therefore, EASEA v0.3 source files can contain a section which will be included *verbatim*.

This offers real freedom to the programmer, who can even include C++ compiler directives in this section.

5 EASEA extensive source file for a simple example

Here is one of the smallest examples of `.ez` source file, implementing the well-known onemax problem (maximising the number of ones in an array of booleans). It is included to show how simple a `.ez` source file looks like. A quick glance shows that EASEA code looks very much like C++, stripped of much of its syntax. C++ style comments are accepted and function bodies are written in pure C++.

As soon as complex type operators (for arrays, lists, trees) already implemented by evolutionary libraries are accepted by EASEA, user-written `initialiser`, `crossover` and `mutator` functions will be superfluous (cf section 7).

```
/*       EASEA implementation of the ONEMAX problem       */

Genome { bool x[15]; }

Standard functions :

Genome::initializer :
  for (int i=0;i<15;i++) Genome.x[i]=tossCoin(.5)?1:0;

Genome::crossover : // Must return the number of concerned children
  int GeneratedChildren=0;
  int CrossoverPosition=(int)random(0,15);
  if (&child1){
    for(int i=0;i<15;i++)
      if (i<CrossoverPosition) child1.x[i]=parent1.x[i];
      else child1.x[i]=parent2.x[i];
    GeneratedChildren++;
  }
  if (&child2){
    for(int i=0;i<15;i++)
      if (i<CrossoverPosition) child2.x[i]=parent2.x[i];
      else child2.x[i]=parent1.x[i];
    GeneratedChildren++;
  }
return GeneratedChildren;

Genome::mutator : // Must return the number of mutations
  int NbMut=0;
  for (int i=0;i<15;i++)
    if (tossCoin(PMut)){
       NbMut++;
       Genome.x[i]=Genome.x[i]?0:1;
    }
  if (NbMut==0) identicalGenome=true;  // saves evaluation time
  return NbMut;

Genome::evaluator : // Must return the score as a float
  float Score=(float)0.0;
  for (int i= 0; i<15;i++)
    Score+=(int)Genome.x[i];
return Score;

Run parameters :
  Population size : 30          // PSize
  Number of generations : 30    // NbGen
  Mutation probability : 0.1    // PMut
  Crossover probability : 1     // PCross
  Genetic engine : SteadyState

End of genome file.
```

To give a rough idea of the volume of code generated by EASEA, `onemax.ez` is 49 lines long, and the generated GALib-compatible `onemax.cpp` file is more than seven times longer with 377 lines.

6 Performance

The concern about performance surfaces whenever a piece of code is generated by a compiler. First of all, as far as syntax is concerned, EASEA produced C++ files are not that different from what human-produced code would have looked like ... after debugging. Semantically speaking, it is true that when writing minor classes, a human programmer will not take the pain of writing code for operators that he knows will never be called. Although such refinement could be included with much pain in EASEA (a first pass could determine which operators of which classes will be needed), the only drawback is that the selection scheme will deal with slightly larger objects than necessary.

However, this cost is negligible, mainly owing to two facts:

1 EASEA generates source code, which is destined to be compiled by an extremely evolved C++ compiler. The code optimisation taking place in the C++ compiler will minimise the lack of optimisation of the EASEA output.

2 EASEA-generated code only concerns the manipulation of genome objects which usually represents only *a few percents* of the total execution time of an evolutionary algorithm (usually overwhelmingly used up by the user-written evaluation function).

7 Future work

Feedback from scientific users is already very positive and shows that v1.0 is still far ahead. Necessary improvements include :

1 support of other libraries (among which EO),

2 utilisation of host libraries complex types and operators (arrays, lists, trees, ... and their corresponding operators),

3 implementation of default representation-independent operators for user-defined genomes,

4 ability to allow user-defined function calls written in any programming language.

The first point is very important, as supporting at least two different libraries is what will give EASEA independence with reference to evolutionary libraries. This will also guide the evolution of the EASEA language towards the really abstract evolutionary programming language it aims to be.

The second point will drastically simplify EASEA source files: most evolutionary libraries already offer complex structures (arrays, lists, trees, ...) and their corresponding operators. As soon as EASEA is capable of making use of those complex types and their default operators, default initialisation, mutation and crossover functions will not be needed anymore in `.ez` files, unless the programmer feels the need to specialise some of them.

There is another way of removing genome-specific operators from `.ez` files: in many cases, user genomes will be aggregates of available types (e.g.: vectors of structures made of floats, integer and symbolic components). It is thus possible to define default operators for such representations using Radcliffe's ideas [7]. The three crossover operators (Random Respectful Recombination, Random Assorting Recombination and Random Transmitting Recombination) as well as the Binomial Minimal Mutation are perfect candidates for that. Of course, representation-specific operators will still be allowed in `.ez` files, as it is acknowledged that they are often more efficient than representation-independent operators [8]. Nevertheless, providing yet efficient default operators will be an important step towards real newcomers in the field (e.g.: "I only want to evolve my vector of structures and don't want to hear about it in the final result").

The last point is equally important to scientific users: many already have their own extremely complex evaluation functions, painstakingly written in FORTRAN or some other language. Heterogeneous function calls could allow them to re-use such evaluation functions, or even plug a hardware device onto the computer which would return a physical evaluation of parameters contained in a genome.

8 Conclusion

Many important fields in computer science have their specific languages (FORTRAN, C/C++, LISP, PROLOG, SMALLTALK, ...). Even complex applications such as databases or spread-sheets have developed their own language ! EA programmers remain however with C++, an inadapted and difficult to use low-level object language. As a result, many scientists have no other choice than wasting a lot of time with becoming computer programmers and rewriting their own evolutionary algorithms. Due to thoroughly different programming techniques and languages, their programs are barely comparable, which is a great obstacle to scientific cooperation and emulation.

The simplicity of EASEA programming is demonstrated with the source code for the onemax problem in section 5. Although the EASEA v0.3 compiler is still minimal (it should not be necessary, for instance, to rewrite completely initialisation, crossover and mutation functions for as basic a structure as an array of booleans) v0.3 can handle linked lists, trees or much more complex structures while hiding from the end-user all of the obscure uninteresting code necessary to operate object-oriented libraries.

EASEA source files are designed to be recompilable with minimal effort on different libraries, so that different research teams will be able to try out each others' implementations in their own environment.

We hope that EASEA will be able to offer the scientific community the means to try out evolutionary algorithms with a minimal time investment as far as programming is concerned. The EASEA v0.3 compiler and its manual are available on the net [2].

References

[1] EVONET *home page:* `http://www.evonet.polytechnique.fr`.

[2] E. Lutton *et al.*, *EVO-Lab home page (EASEA v0.3):* `http://www-rocq.inria.fr/EVO-Lab/`.

[3] J. J. Merelo, *EO home page:* `http:/fast.to/EO`, Granada University.

[4] P. Stearns, *ALex & AYacc home page:* `http://www.bumblebeesoftware.com`, Bumblebee Software Ltd.

[5] M. Wall, *GAlib home page:* `http://www.mit.edu/people/moriken/doc/galib`, MIT.

[6] I. Landrieu, B. Naudts, "An Object Model for Search Spaces and their Transformations," Artificial Evolution conference, EA'99, 3-5 Nov 99, Dunkerque, France, 1999.

[7] N. J. Radcliffe, "Forma Analysis and Random Respectful Recombination," ICGA'91, proceedings pp222-229, 1991.

[8] N. J. Radcliffe and P. D. Surry, "Fitness variance of formae and performance prediction," FOGA'95, pp51-72, Morgan Kaufmann publ., 1995.

[9] P. D. Surry and N. J. Radcliffe, "Formal Algorithms + Formal Representation = Search Strategies," PPSN'96, proceedings 1141 pp366-375, 1996.

[10] P. D. Surry, "A Prescriptive Formalism for Constructing Domain-Specific Evolutionary Algorithms," PhD thesis, University of Edinburgh, 1998.